CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 27: AVL Trees

Announcements

- PA2 due Sunday (last day to submit with grace days/penalty is Tues)
- WA4 coming soon
- Midterm #2 on Friday

BST Operations

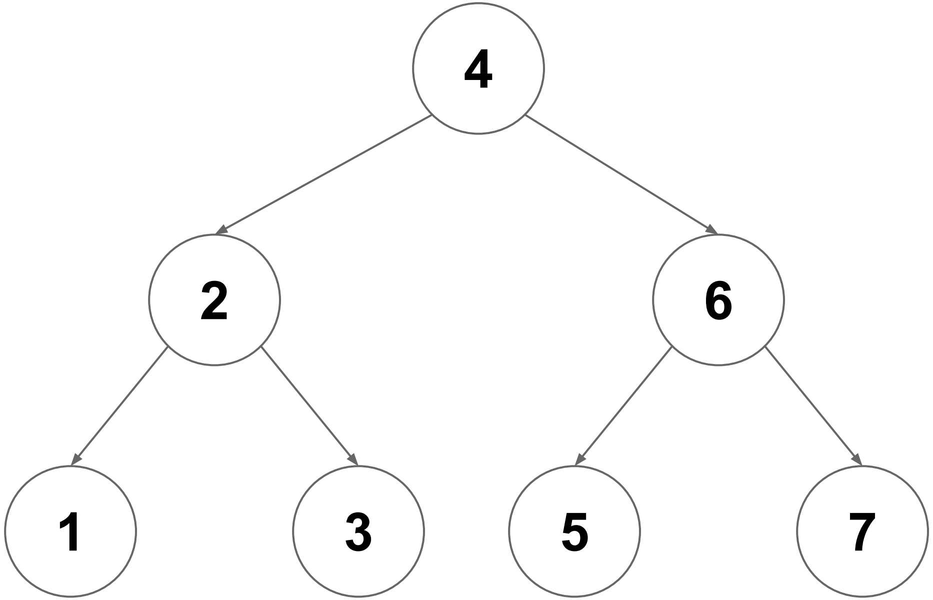
Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?* $O(n)$

$\log(n) \le d \le n$

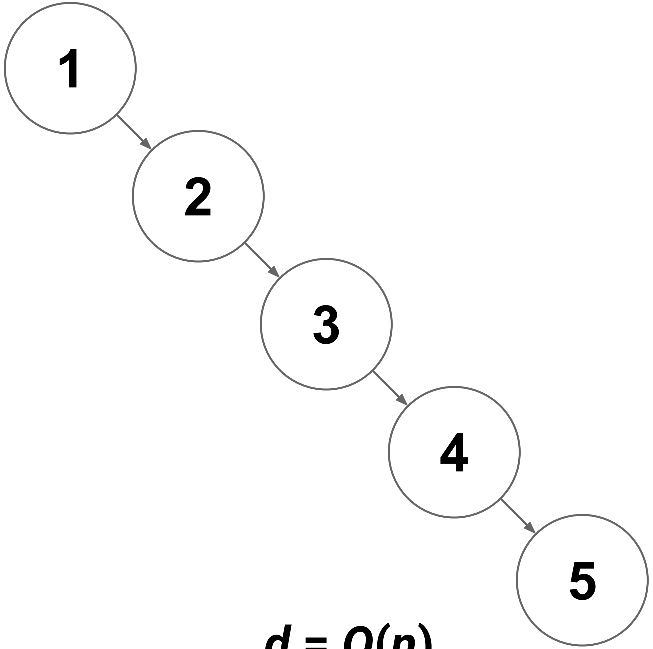
Tree Depth vs Size

If height(left) ≈ height(right)



$d = O(\log(n))$

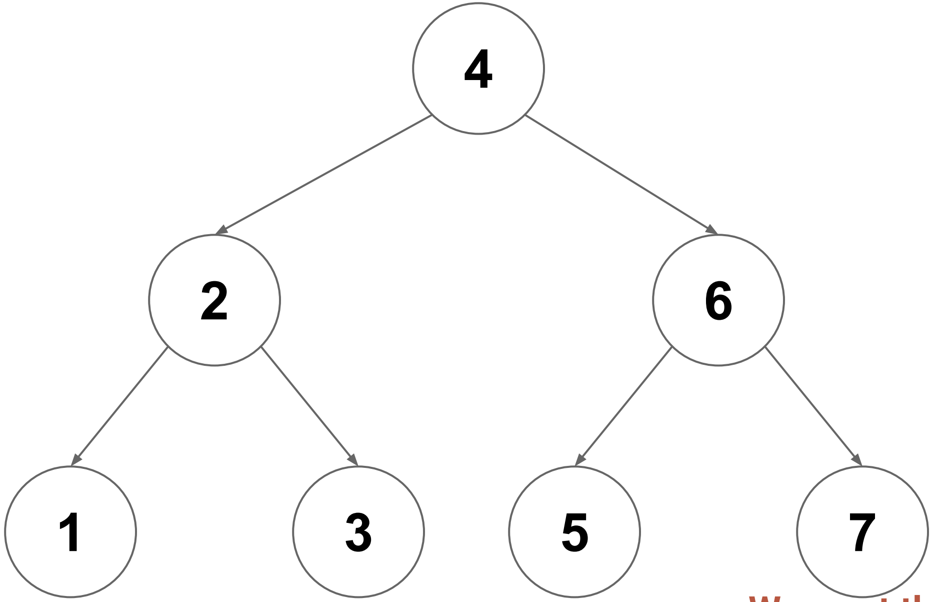
If height(left) ≪ height(right)



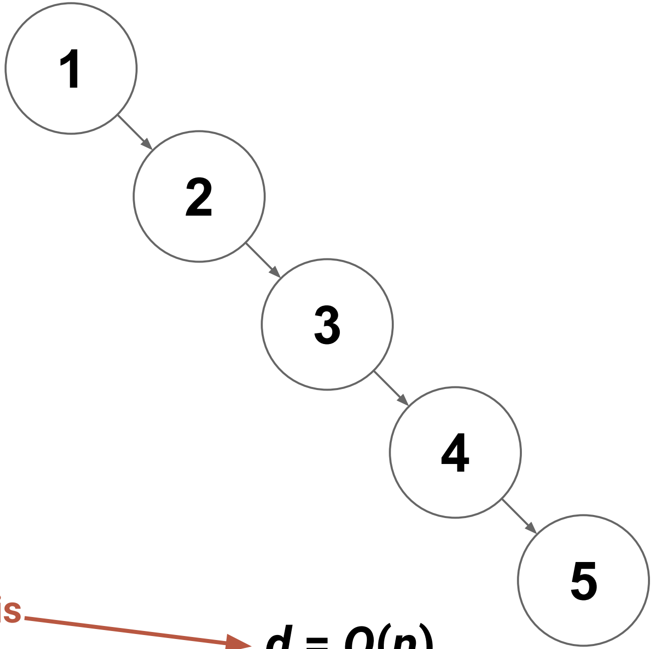
$d = O(n)$

Tree Depth vs Size

If height(left) ≈ height(right)



If height(left) ≪ height(right)



We want this, not this

$d = O(\log(n))$ $d = O(n)$

Keeping Depth Small - Two Approaches

Option 1

Keep tree **balanced**: subtrees **+/-1** of each other in height

(add a field to track amount of "imbalance")

Option 2

Keep leaves at some minimum depth (***d*/2**)

(Add a color to each node marking it as "red" or "black")

Balanced Trees

Balanced Trees are good: Faster `find`, `insert`, `remove`

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

What do we mean by balanced?

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

What do we mean by balanced? **|height(right) - height(left)| ≤ 1**

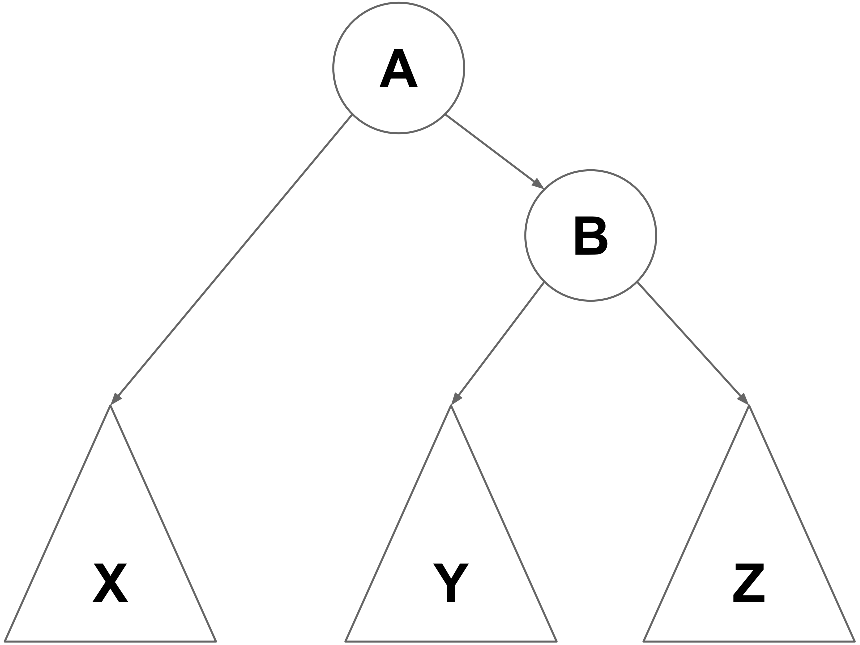
Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

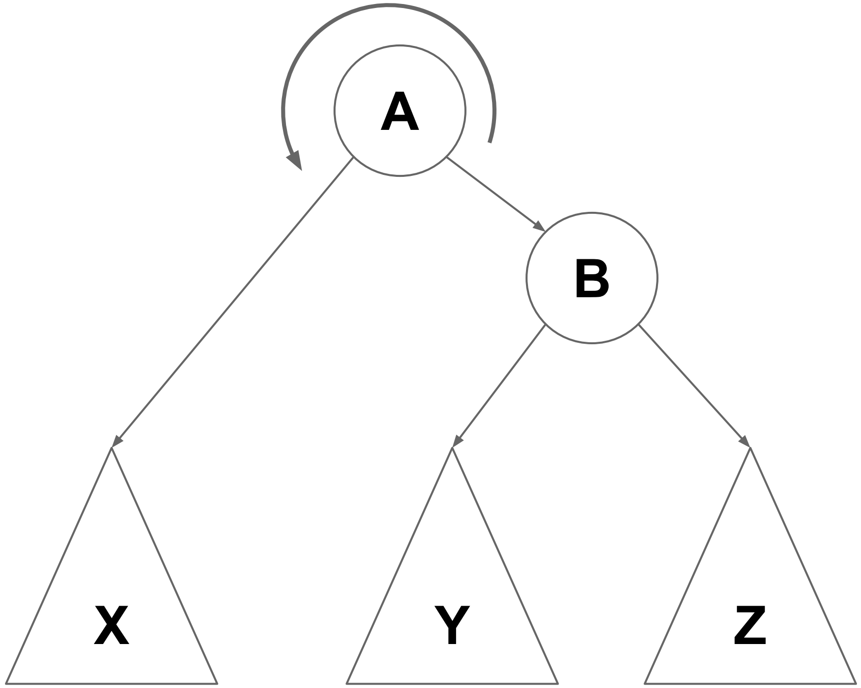
What do we mean by balanced? **|height(right) - height(left)| ≤ 1**

How do we keep a tree balanced?

Rebalancing Trees (rotations)

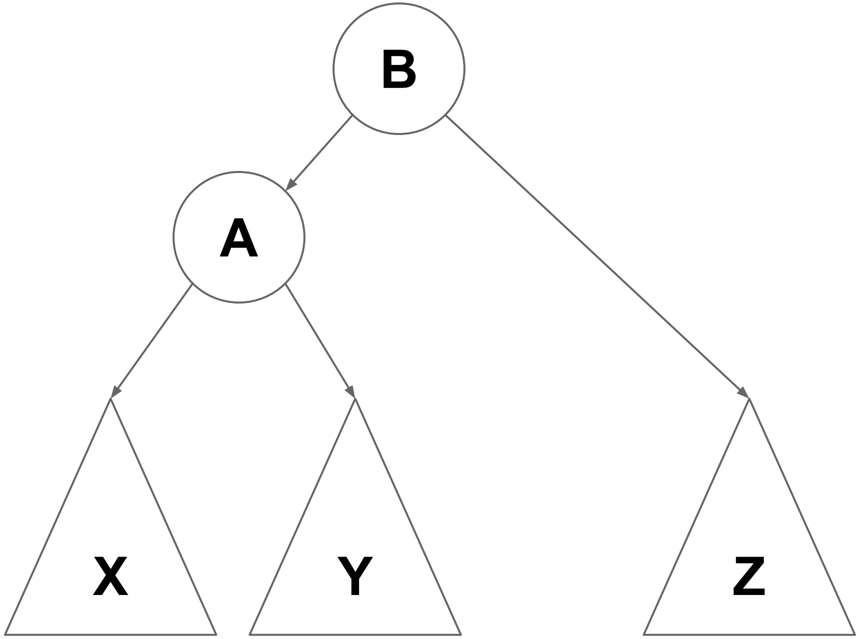


Rebalancing Trees (rotations)



Rotate(A, B)

Rebalancing Trees (rotations)

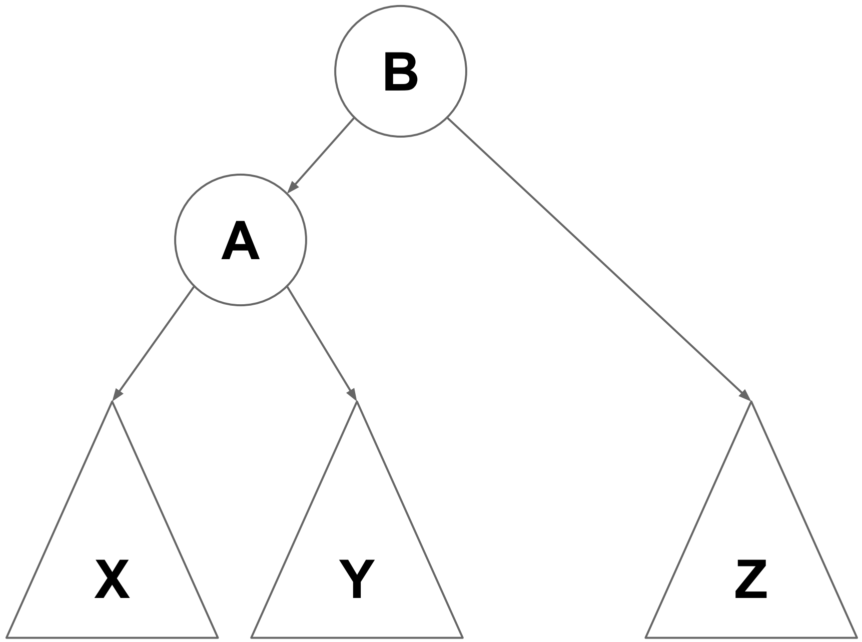


Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child



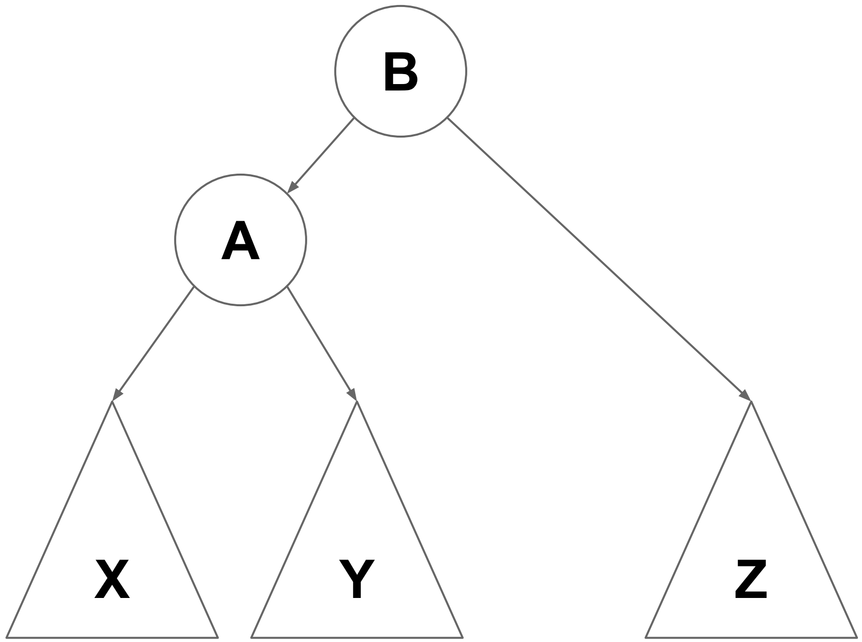
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained?



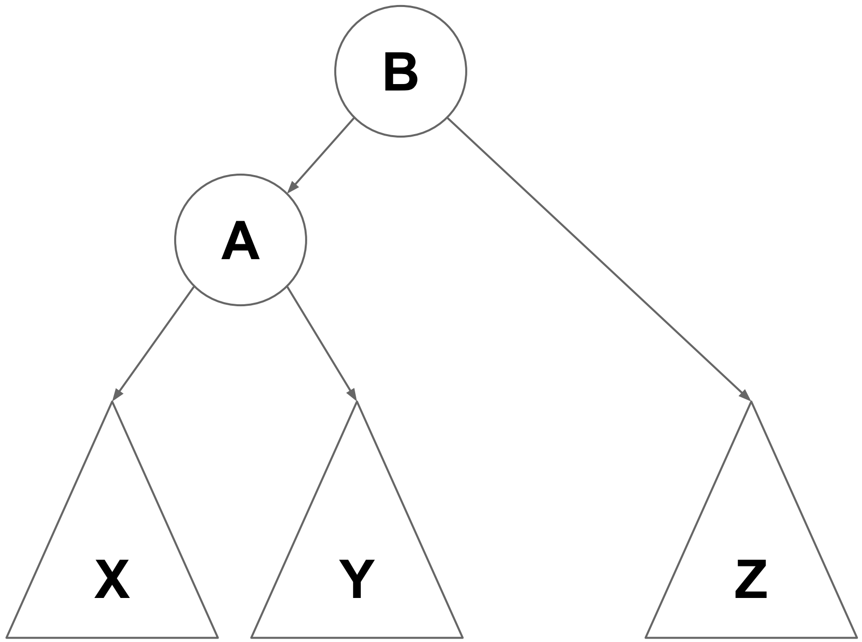
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**



Rotate(A, B)

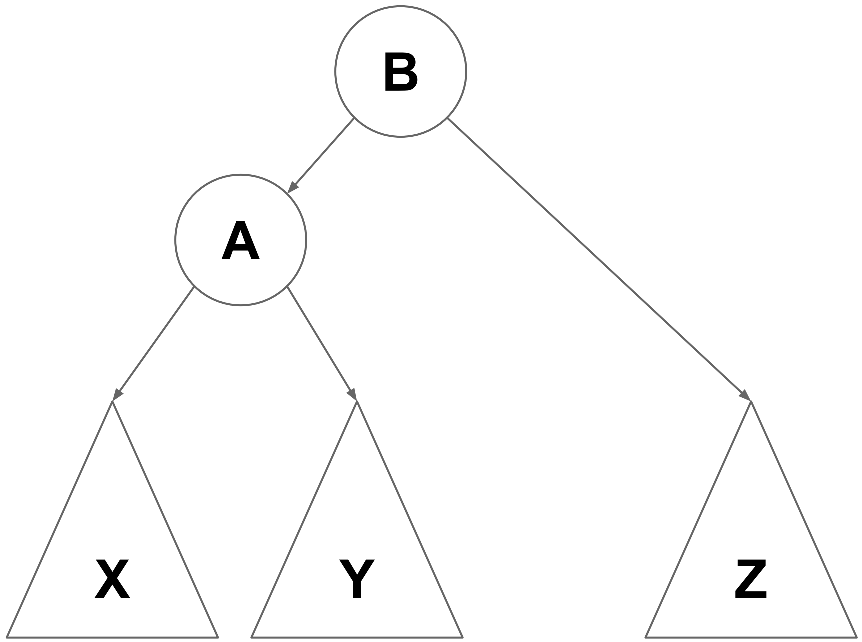
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity?



Rotate(A, B)

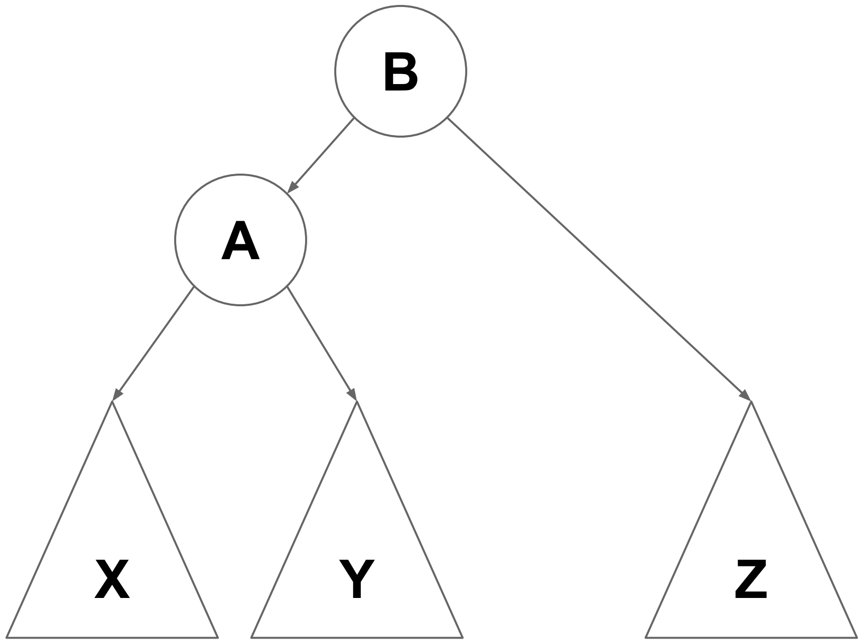
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity? ***O*(1)**



Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

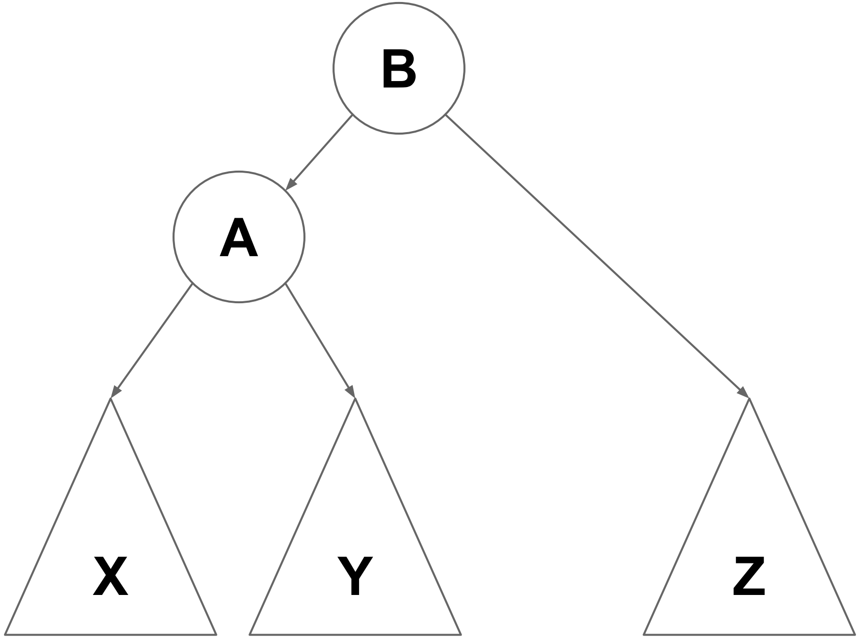
B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity? ***O*(1)**

This is called a left rotation

(right rotation is the opposite)



Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

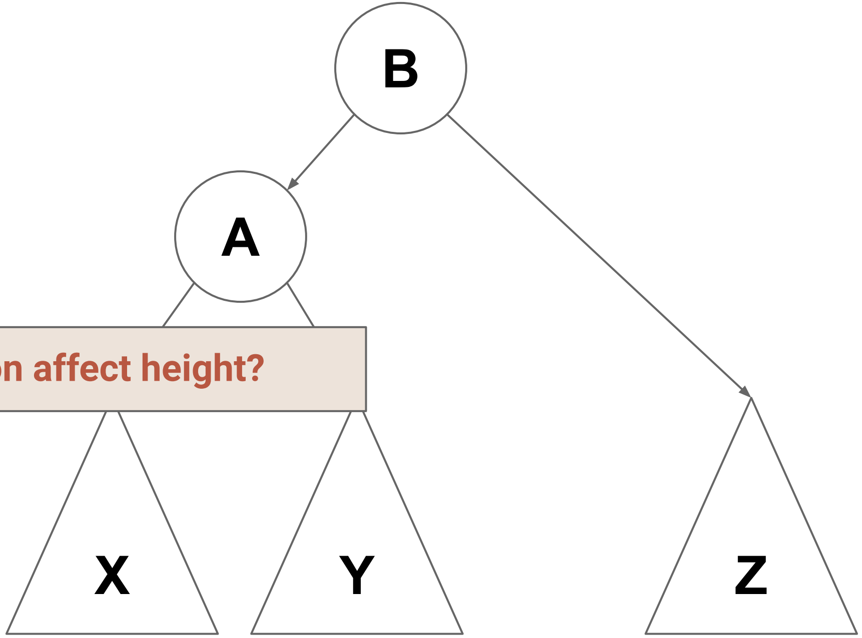
Is ordering maintained? **Yes!**

Complexity? ***O*(1)**

This is called a left rotation

(right rotation is the opposite)

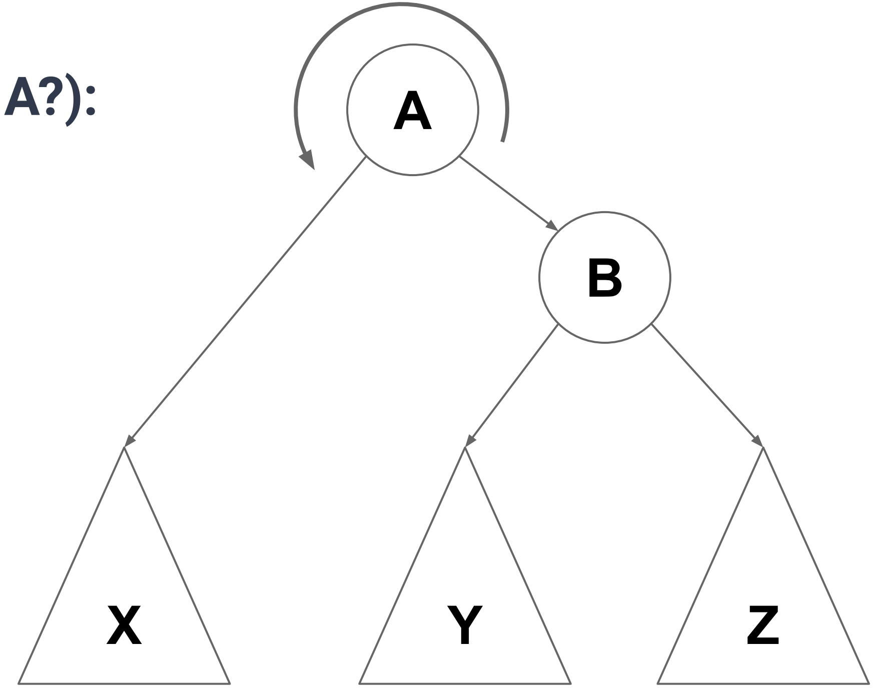
How does a rotation affect height?



Rotate(A, B)

Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):

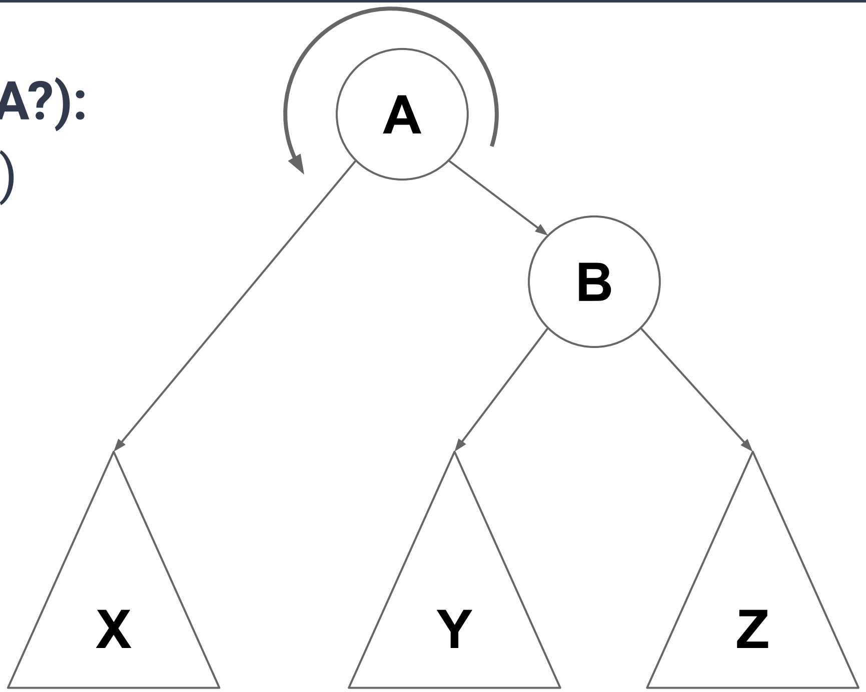


Rotate(A, B)

Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):

h(A) = 1 + max(h(X), 1 + max(h(Y), h(Z))



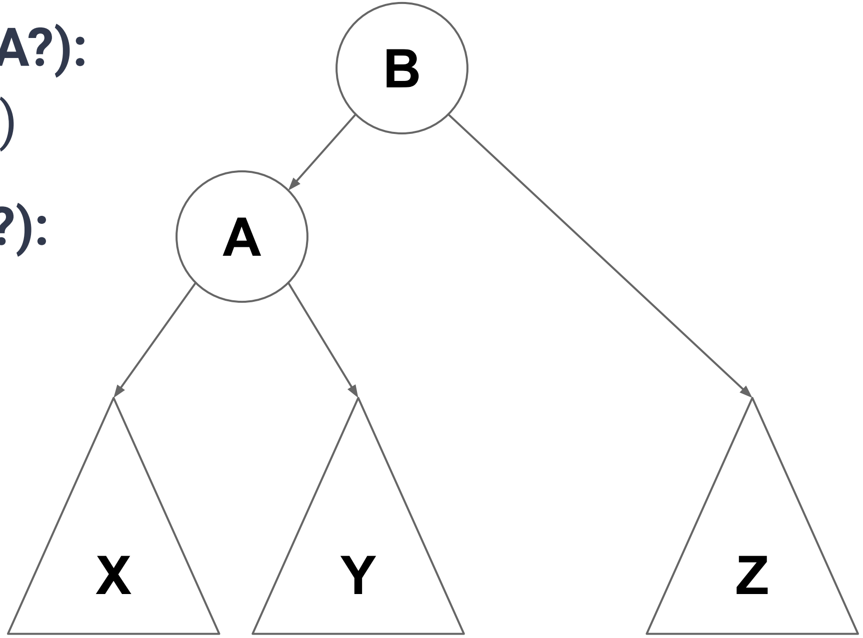
Rotate(A, B)

Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):

h(A) = 1 + max(h(X), 1 + max(h(Y), h(Z)))

After Rotation (what is the height of B?):



Rotate(A, B)

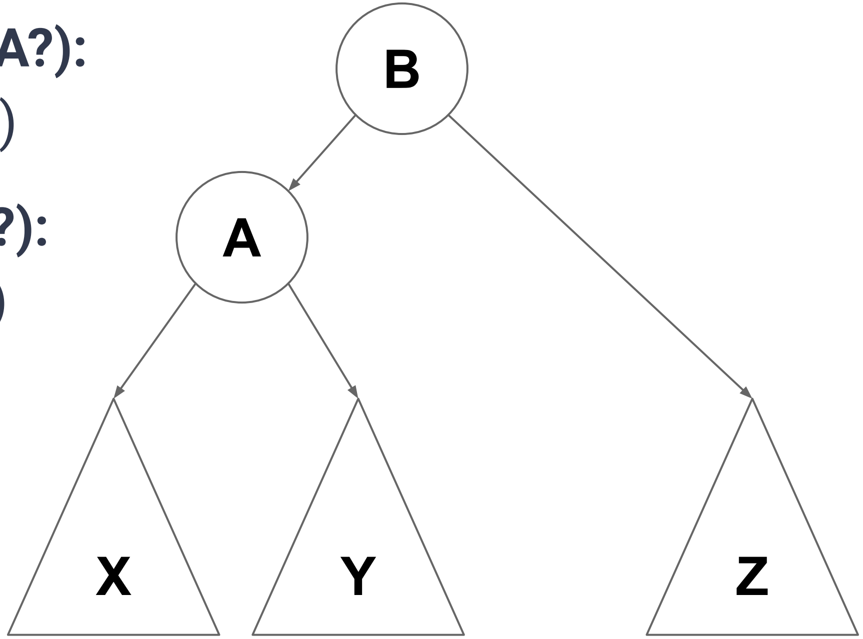
Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):

h(A) = 1 + max(h(X), 1 + max(h(Y), h(Z))

After Rotation (what is the height of B?):

h(B) = 1+ max(1 + max(h(X),h(Y)), h(Z))



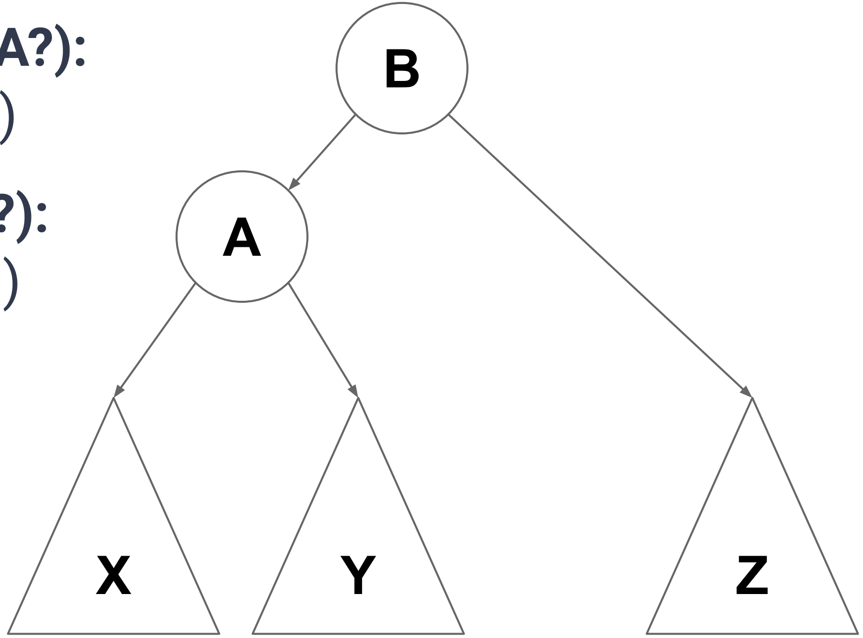
Rotate(A, B)

Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):
h(A) = 1 + max(h(X), 1 + max(h(Y), h(Z))

After Rotation (what is the height of B?):
h(B) = 1 + max(1 + max(h(X),h(Y)), h(Z))

- If **X** was the tallest of **X,Y,Z** our total height increased by 1.
- If **Z** was the tallest our total height decreased by 1.
- If **X,Z** same height, or **Y** is the tallest then total is unchanged



Rotate(A, B)

Rebalancing Trees (rotations)

Before Rotation (what is the height of A?):

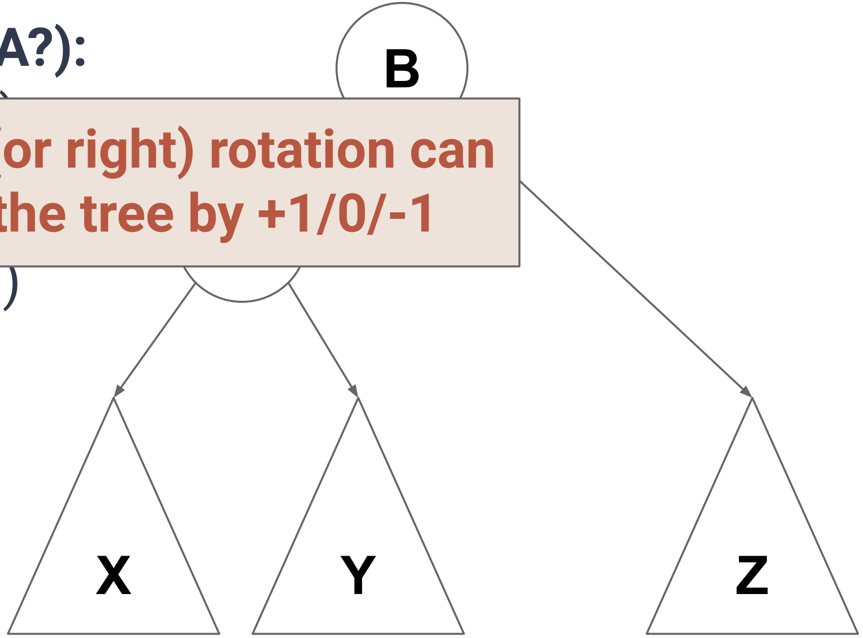
h(A) = 1 + ma

After Rotatio

h(B) = 1 + max(1 + max(h(X),h(Y)), h(Z))

Therefore, a single left (or right) rotation can change the height of the tree by +1/0/-1

- If **X** was the tallest of **X,Y,Z** our total height increased by 1.
- If **Z** was the tallest our total height decreased by 1.
- If **X,Z** same height, or **Y** is the tallest then total is unchanged



Rotate(A, B)

AVL Trees

AVL Trees

An **AVL tree** (**A**delson-**V**elsky and **L**andis) is a ***BST*** where every subtree is depth-balanced

Remember: Tree depth = height(root)

Balanced: |height(root.right) - height(root.left)| ≤ 1

AVL Trees

Define **`balance(`*`v`*`) = height(`*`v.right`*`) - height(`*`v.left`*`)`**

Goal: Maintaining **`balance(`*`v`*`)` ∈ { -1, 0, 1 }**

- **`balance(`*`v`*`) = 0`** → "***v*** is balanced"
- **`balance(`*`v`*`) = -1`** → "***v*** is left-heavy"
- **`balance(`*`v`*`) = 1`** → "***v*** is right-heavy"

AVL Trees

Define **`balance(`*`v`*`) = height(`*`v.right`*`) - height(`*`v.left`*`)`**

Goal: Maintaining **`balance(`*`v`*`)`** $\in \{-1, 0, 1\}$

- **`balance(`*`v`*`) = 0`** → "***v*** is balanced"
- **`balance(`*`v`*`) = -1`** → "***v*** is left-heavy"
- **`balance(`*`v`*`) = 1`** → "***v*** is right-heavy"

What does enforcing this gain us?

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of **log(*n*)**

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

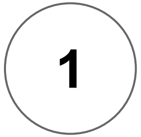
YES! Depth balance forces a maximum possible depth of **log(*n*)**

Proof Idea: An AVL tree with depth ***d*** has "enough" nodes

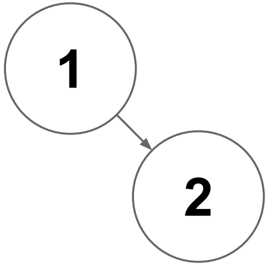
AVL Trees - Depth Bounds

Let `minNodes(d)` be the min number of nodes an in AVL tree of depth ***d***

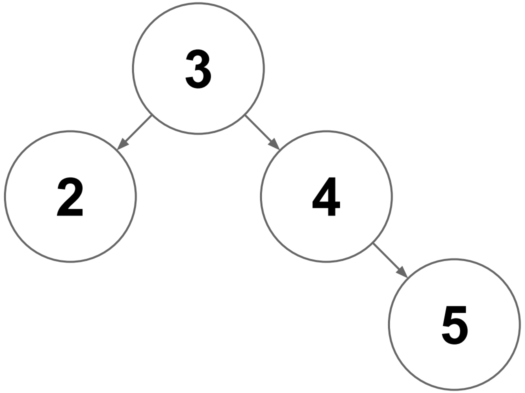
minNodes(0) = 1



minNodes(1) = 2

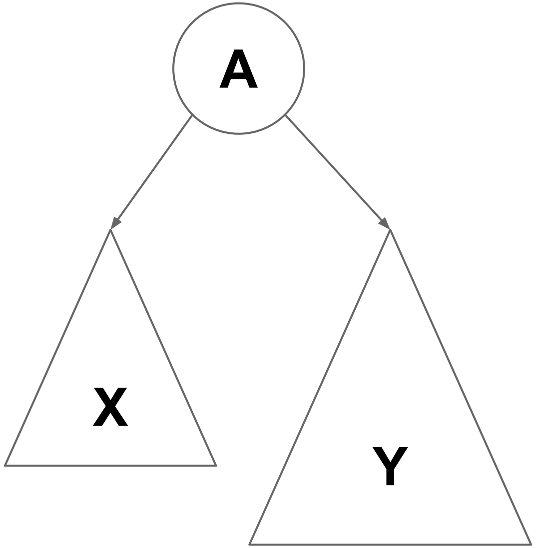


minNodes(2) = 4



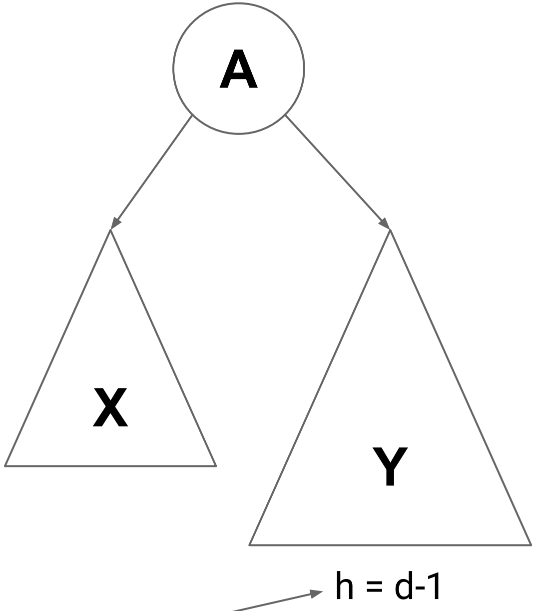
AVL Trees - Depth Bounds

For any tree of depth ***d***:



AVL Trees - Depth Bounds

For any tree of depth ***d***:

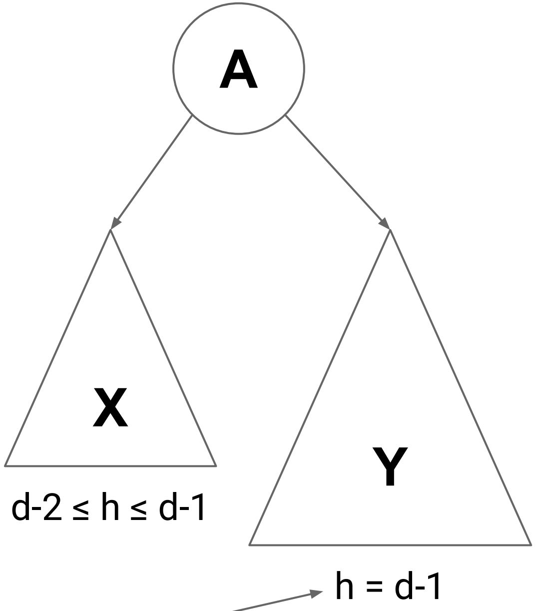


At least one subtree must have depth of ***d* - 1**
(because total depth is ***d***)

AVL Trees - Depth Bounds

For any tree of depth ***d***:

The other subtree must have a depth of at least ***d* - 2** because the AVL constraint does not allow it to differ by more than 1



At least one subtree must have depth of ***d* - 1** (because total depth is ***d***)

AVL Tree - Depth Bounds

For ***d*** < 1:

minNodes(***d***) = 1 + minNodes(***d* - 1**) + minNodes(***d* - 2**)

AVL Tree - Depth Bounds

For $\boldsymbol{d} < 1$:

$\text{minNodes}(\boldsymbol{d}) = 1 + \boxed{\text{minNodes}(\boldsymbol{d} \mathbf{- 1}) + \text{minNodes}(\boldsymbol{d} \mathbf{- 2})}$

This is the Fibonacci Sequence!

AVL Tree - Depth Bounds

For $d < 1$:

minNodes(d) = 1 + $\boxed{\text{minNodes}(d - 1) + \text{minNodes}(d - 2)}$

This is the Fibonacci Sequence!

What is the d^{th} term of the Fibonacci sequence?

Coarse approximation: minNodes(d) = $\Omega(1.5^d)$

https://en.wikipedia.org/wiki/Fibonacci_sequence

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^{d})$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^{d})$

$$n \geq c1.5^d$$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^{d})$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^{d})$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^{d})$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

AVL Tree - Depth Bounds

minNodes(***d***) = Ω(1.5^{d})

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

$$\frac{\log_2(n))}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^d)$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

$$\frac{\log_2(n))}{\boxed{\log_2(1.5)}} - \boxed{\frac{\log_2(c)}{\log_2(1.5)}} \geq d$$

All constants

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^d)$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2(1.5^d)$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

$$\frac{\log_2(n))}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$$

$$d \in O(\log_2(n))$$

AVL Tree - Depth Bounds

minNodes(***d***) = **Ω**(1.5^{d})

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \quad \geq d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log$$

Therefore if we enforce the AVL constraint, then a tree with *n* nodes will have logarithmic depth

$$\frac{\quad}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$d \in O(\log_2(n))$$

AVL Tree - Depth Bounds

minNodes(***d***) = $\Omega(1.5^d)$

$$n \geq c1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\log_2\left(\frac{n}{c}\right) \quad \geq d$$

$$\frac{\ldots g_2(c)}{\ldots_2(1.5)} \geq d$$

$$d \in O(\log_2(n))$$

Therefore if we enforce the AVL constraint, then a tree with *n* nodes will have logarithmic depth

So how do we enforce the constraint?

Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node

Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node

Idea: Store height of each node at the node

Enforcing the AVL Constraint

- Computing `balance()` on the fly is expensive
  - `balance()` calls `height()` twice
  - Computing `height()` requires visiting every node

Idea: Store height of each node at the node

Better Idea: Just store the balance factor (only needs 2 bits)

Enforcing the AVL Constraint

```
public class AVLTreeNode<T> {
  T value;
  Optional<AVLTreeNode<T>> parent; // We need a ref to parent to rotate
  Optional<AVLTreeNode<T>> leftChild;
  Optional<AVLTreeNode<T>> rightChild;
  Boolean isLeftHeavy;   // true if height(right) - height(left) == -1
  Boolean isRightHeavy;  // true if height(right) - height(left) == 1
}
```

Need to add 3 fields to our TreeNode class to make it an AVLTreeNode

Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of `insert`?

Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of `insert`? Increases by **at most** 1

Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of `insert`? Increases by **at most** 1
- What is the effect on the height of `remove`?

Enforcing the AVL Constraint

Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of `insert`? Increases by **at most** 1
- What is the effect on the height of `remove`? Decreases by **at most** 1

Enforcing the AVL Constraint

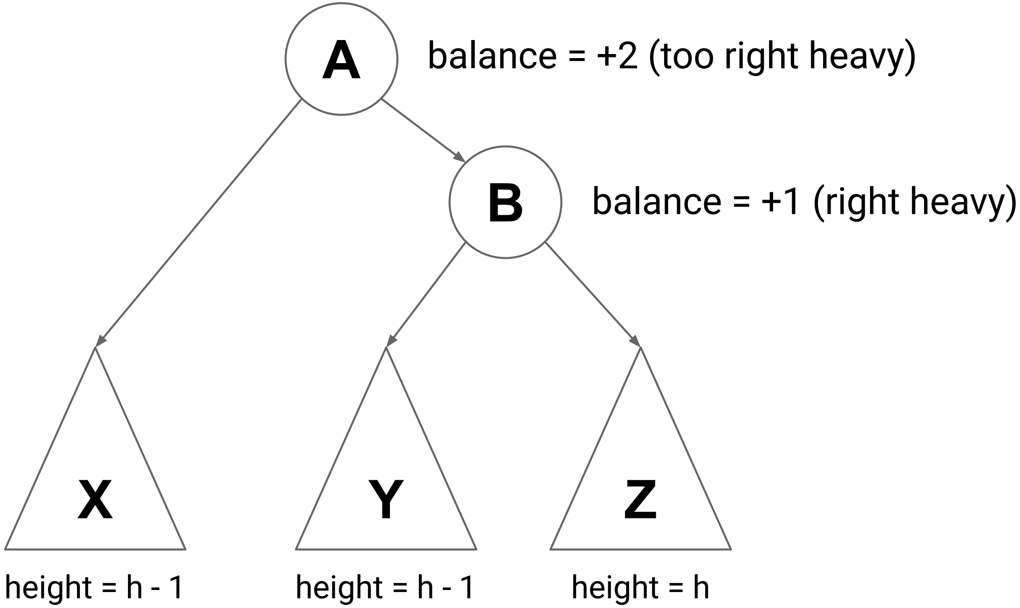
Assume we have a valid AVL tree and we modify it. How might this break our AVL constraint?

- What is the effect on the height of `insert`? Increases by **at most** 1
- What is the effect on the height of `remove`? Decreases by **at most** 1

Therefore after an operation that modifies an AVL tree, the difference in heights can be **at most** 2.

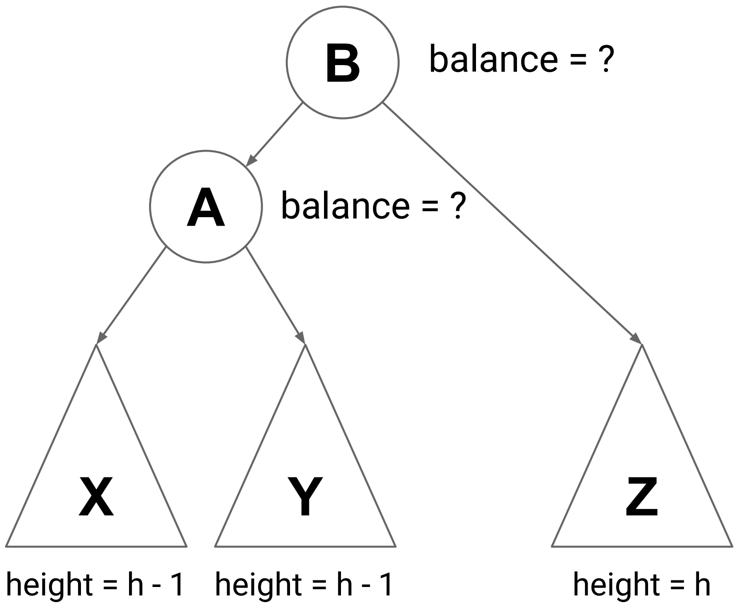
What are the exact ways this broken constraint might show up?

Enforcing the AVL Constraint: Case 1



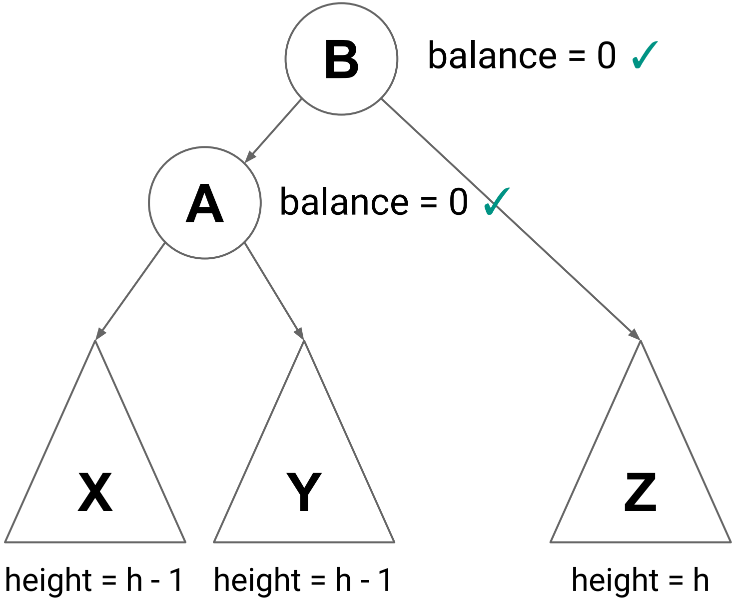
How can we fix this?

Enforcing the AVL Constraint: Case 1



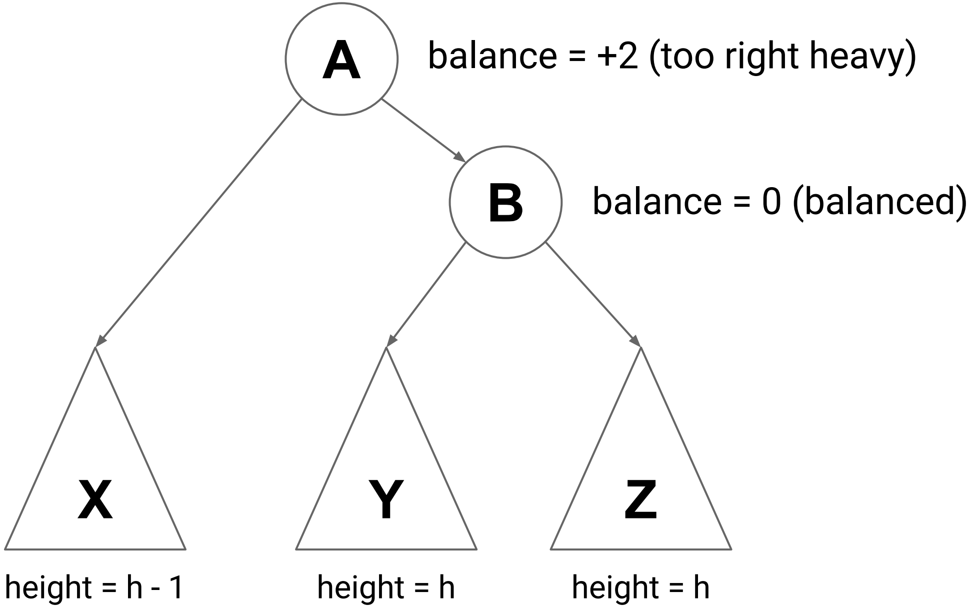
How can we fix this? **`rotate(A,B)`**

Enforcing the AVL Constraint: Case 1



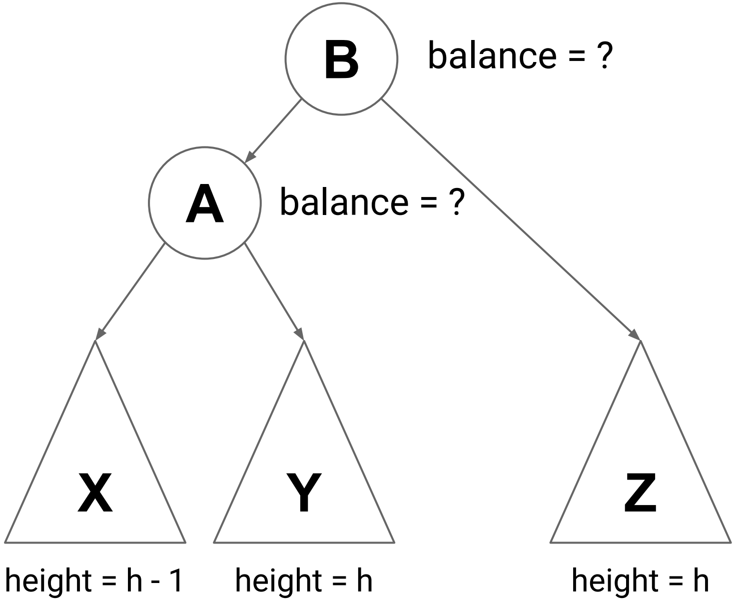
How can we fix this? **`rotate(A,B)`**

Enforcing the AVL Constraint: Case 2



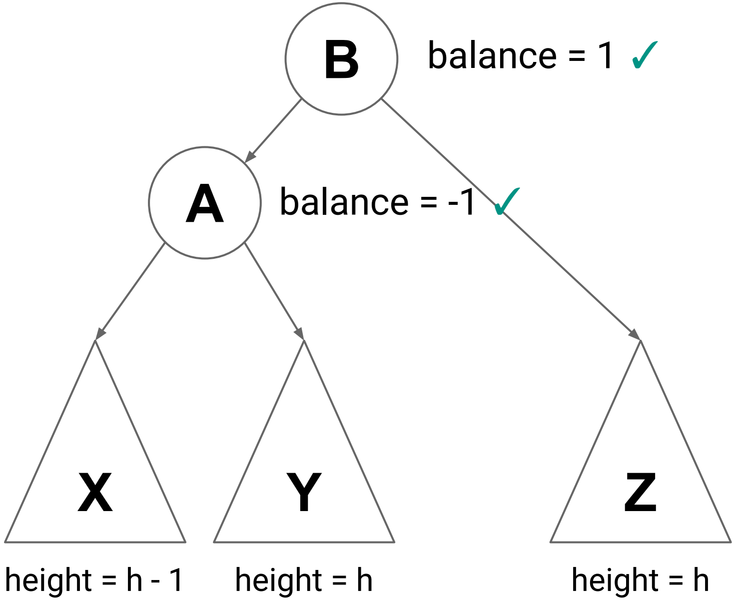
How can we fix this?

Enforcing the AVL Constraint: Case 2



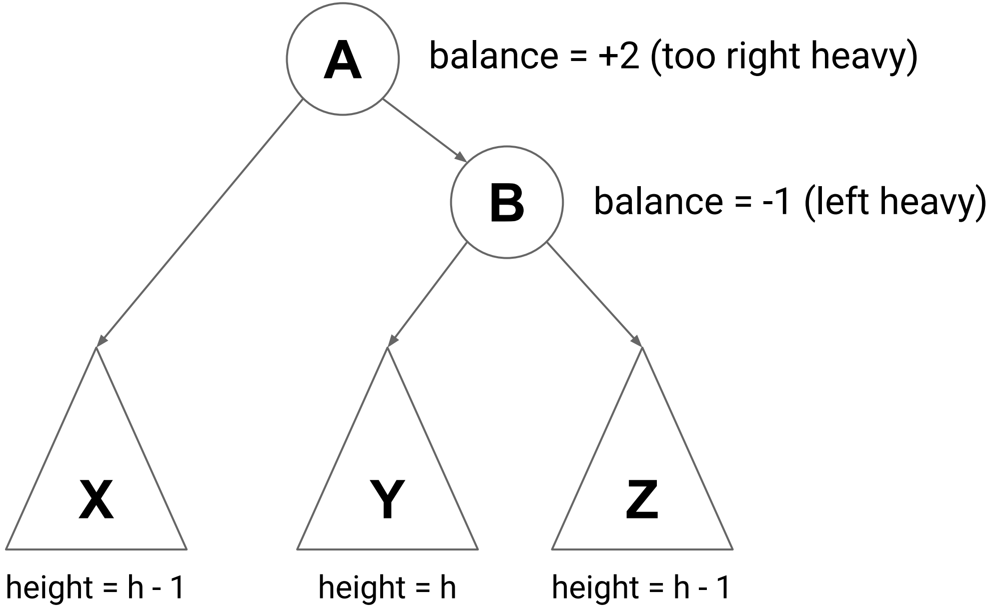
How can we fix this? **`rotate(A,B)`**

Enforcing the AVL Constraint: Case 2



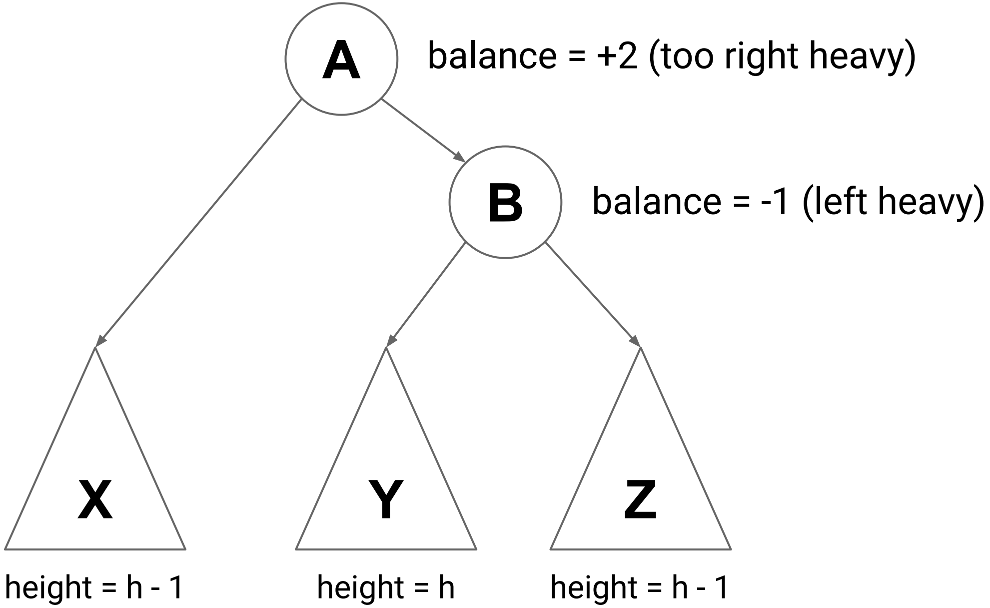
How can we fix this? **`rotate(A,B)`**

Enforcing the AVL Constraint: Case 3



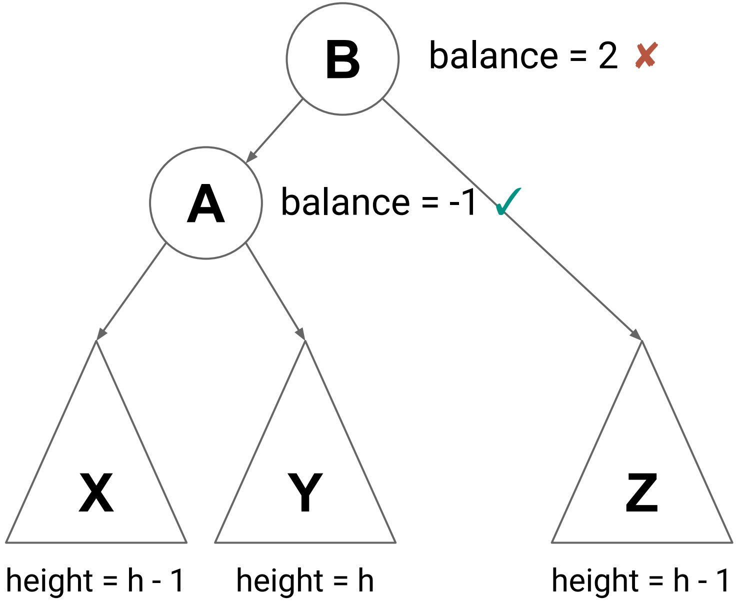
How can we fix this?

Enforcing the AVL Constraint: Case 3



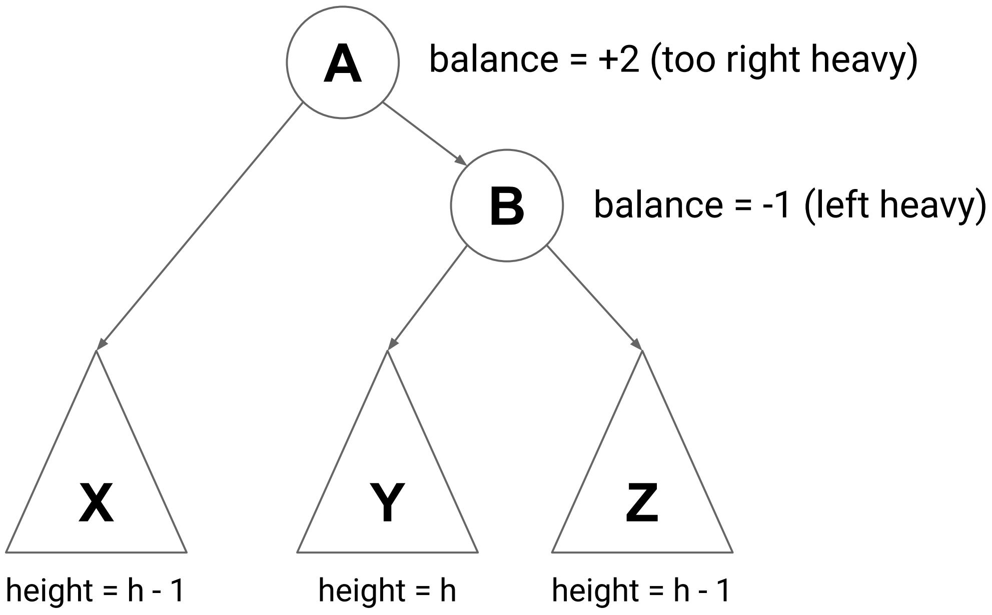
How can we fix this?
Will just a single left rotation work?

Enforcing the AVL Constraint: Case 3



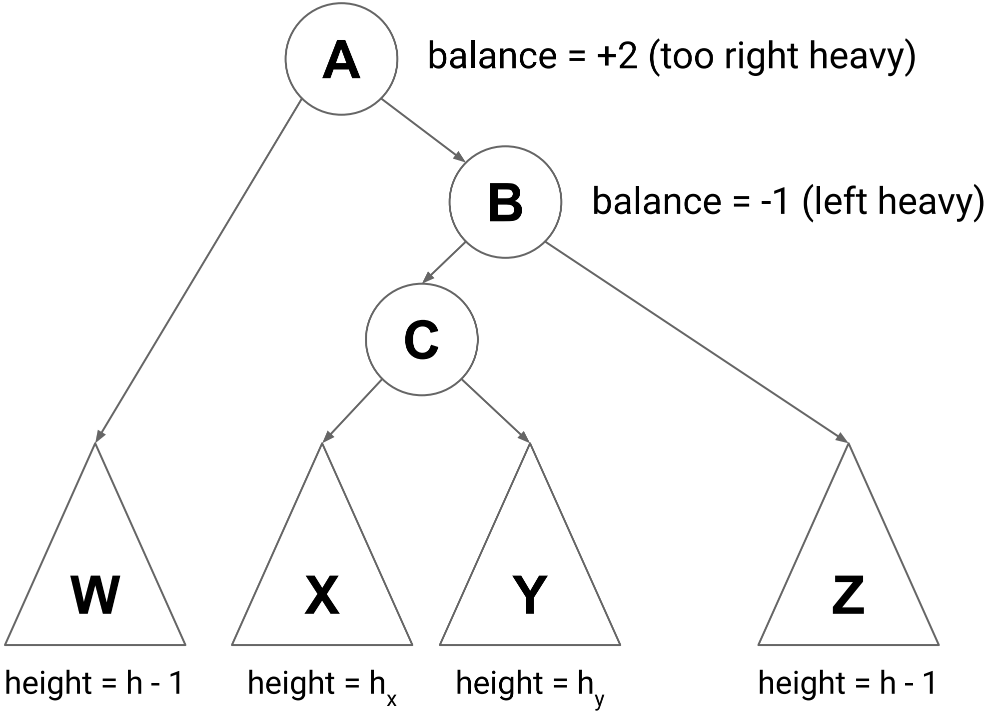
How can we fix this?
Will just a single left rotation work? **No**

Enforcing the AVL Constraint: Case 3



How can we fix this?

Enforcing the AVL Constraint: Case 3



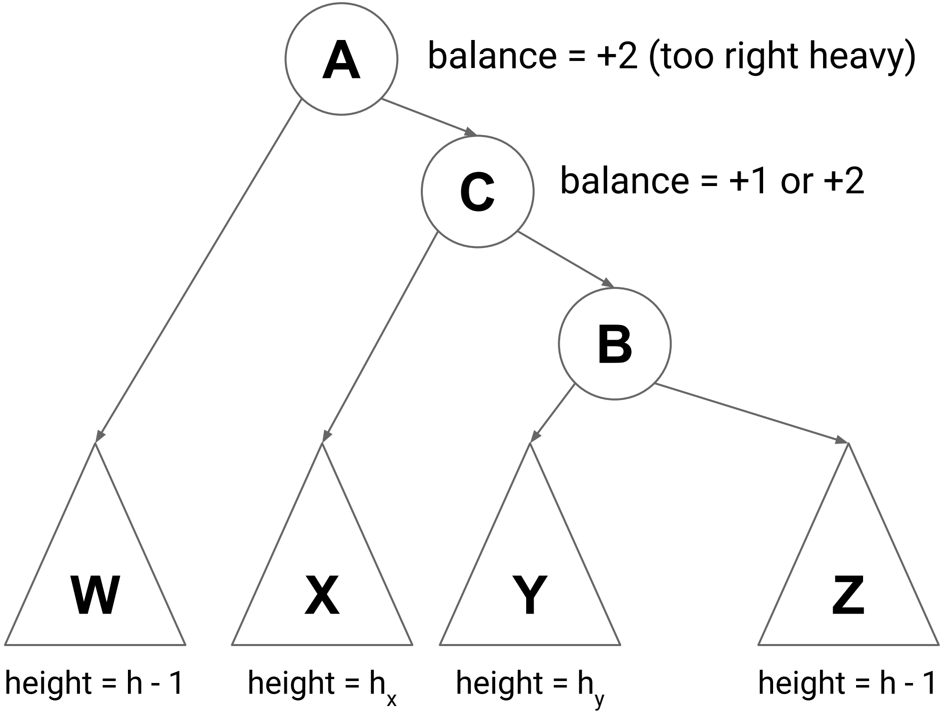
How can we fix this?

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_x}$ or $\boldsymbol{h_y}$ must be ***h* - 1**

The other can also be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint: Case 3



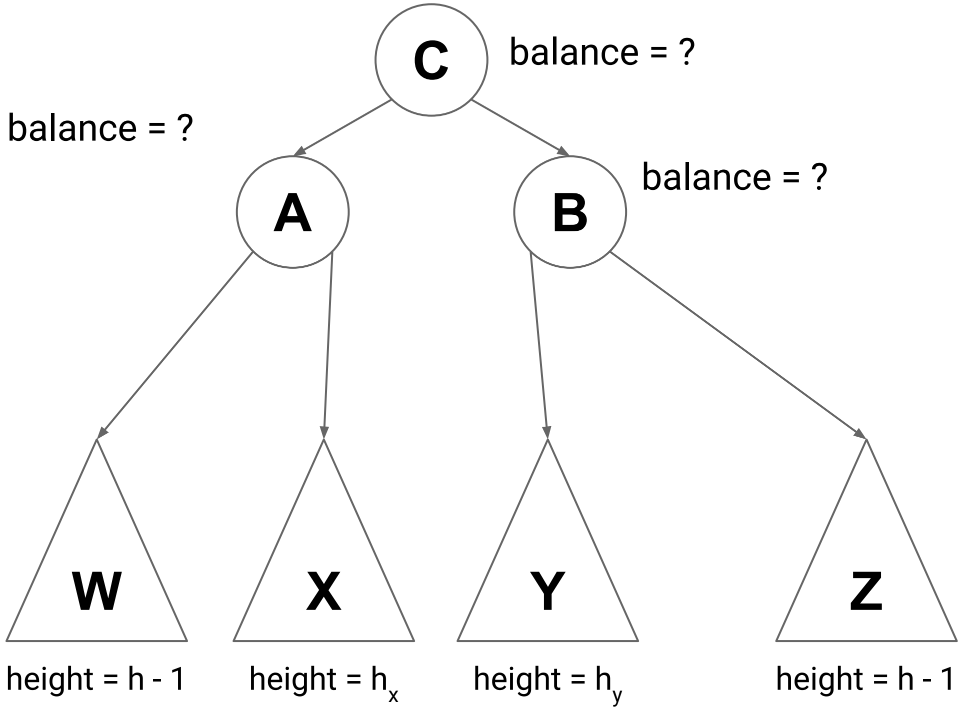
How can we fix this?
Rotate right first: **`rotate(B,C)`**

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_x}$ or $\boldsymbol{h_y}$ must be ***h* - 1**

The other can also be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint: Case 3



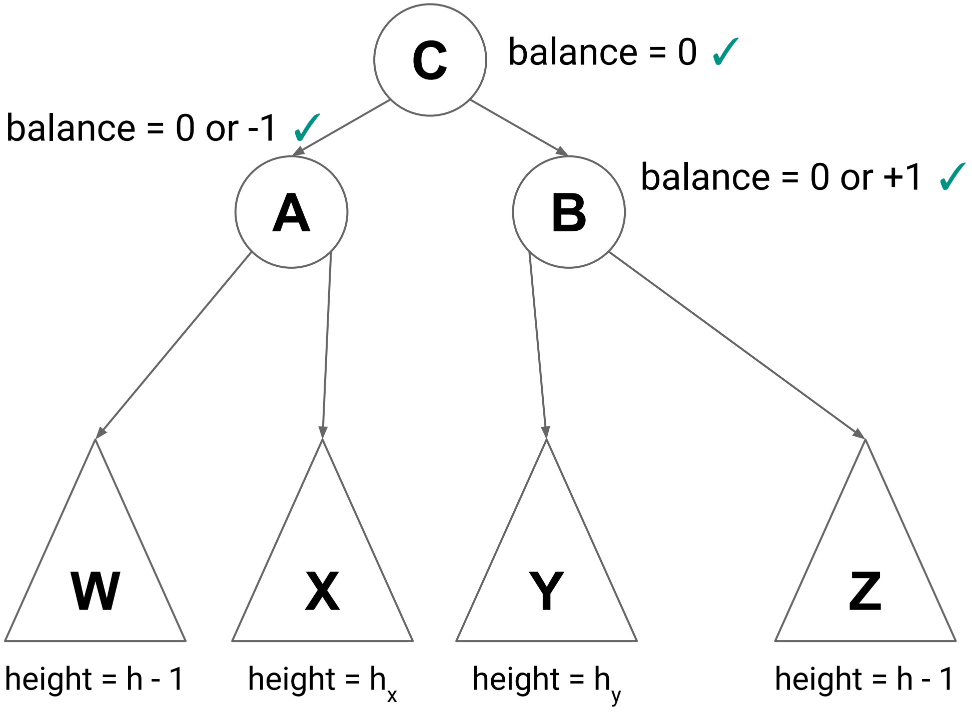
How can we fix this?
Rotate right first: **`rotate(B,C)`**
Then right left: **`rotate(A,C)`**

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_x}$ or $\boldsymbol{h_y}$ must be ***h* - 1**

The other can also be ***h* - 2**, or ***h* - 1**

Enforcing the AVL Constraint: Case 3



How can we fix this?
Rotate right first: **`rotate(B,C)`**
Then right left: **`rotate(A,C)`**

Height of **C** we know must be ***h***

Therefore At least one of $\boldsymbol{h_x}$ or $\boldsymbol{h_y}$ must be ***h*** **- 1**

The other can also be ***h*** **- 2**, or ***h*** **- 1**

Enforcing the AVL Constraint

- If too right heavy (balance == +2)
  - If right child is right heavy (balance == +1) or balanced (balance == 0)
    - rotate left around the root
  - If right child is left heavy (balance == -1)
    - rotate right around root of right child, then rotate left around root
- If too left heavy (balance == -2)
  - Same as above but flipped

Therefore if we have a balance factor that is off, but all children are AVL trees, we can fix the balance factor in at most 2 rotations

Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST)
2. Insert the new leaf and set balance factor to 0
3. Trace path back up to root and update balance factors
    a. If a balance factor becomes +/-2 then rotate to fix

Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST) — $\boldsymbol{O(d) = O(\log n)}$
2. Insert the new leaf and set balance factor to 0 — $\boldsymbol{O(1)}$
3. Trace path back up to root and update balance factors — $\boldsymbol{O(d) = O(\log n)}$
    a. If a balance factor becomes +/-2 then rotate to fix — $\boldsymbol{O(1)}$

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Inserting New Nodes

```
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Find insertion point and create the new leaf $O(d) = O(\log n)$

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

$O(d) = O(\log n)$ iterations

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

What is the cost of each iteration?
How exactly do we fix the issues? (next slide)

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a right heavy subtree, then the subtree is no longer right heavy and we can stop here

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null)
  // Fix issues that occur from inserting into
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy) {
    if (newNode.isLeftHeavy) newNode.parent.get().rotateRight();
    else newNode.parent.get().rotateLeftRight();
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a left heavy subtree, then we just created imbalance, and need to rotate. But then we can stop.

Inserting New Nodes

```
if (newNode.parent.get().leftChild.orElse(null) == newNode) {
  // Fix issues that occur from inserting into parents left subtree
  if (newNode.parent.get().isRightHeavy) {
    newNode.parent.get().isRightHeavy = false;
    return
  } else if (newNode.parent.get().isLeftHeavy)
    if (newNode.isLeftHeavy) newNode.parent.get
    else newNode.parent.get().rotateLeftRight()
    return
  } else {
    newNode.parent.get().isLeftHeavy = true;
  }
}
```

If we inserted into the left of a balanced subtree, then we mark it as now being left heavy, and continue up the tree

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

What is the cost of each iteration? **O(1)**

Inserting New Nodes

```java
public void insert(T value, AVLTreeNode<T> root) {
  // Use normal logic for inserting into a BST, then set heavy flags
  AVLTreeNode<T> newNode = insertIntoBST(value, root);
  newNode.isLeftHeavy = newNode.isRightHeavy = false;
  while (newNode.parent.isPresent()) {
    if (newNode.parent.get().leftChild.orElse(null) == newNode) {
      // Fix issues that occur from inserting into parents left subtree
    } else {
      // Fix issues that occur from inserting into parents right subtree
    }
    newNode = newNode.parent.get();
  }
}
```

Therefore, our total insertion cost is **O(d) = O(log(n))**

Removing Records

- Removal follows essentially the same process as insertion
  - Do a normal BST removal
  - Go back up the tree adjusting balance factors
  - If you discover a balance factor that goes to +2/-2, rotate to fix

Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)

Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $\boldsymbol{d = O(\log(n))}$

Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are $|\text{height}(\text{right}) - \text{height}(\text{left})| \leq 1$
  - It will guarantee $\boldsymbol{d = O(\log(n))}$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1

Summary

- We want shallow BSTs (it makes `find`, `insert`, `remove` faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $d = O(\log(n))$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1
- Therefore after `insert`/`remove` into an AVL tree, we can reinforce AVL constraints with one (or two) rotations
  - We only need to make one trip back up the tree to do so
  - Therefore `insert`/`remove` is still $O(d) = O(\log(n))$